CS 261-020
Data Structures

Lecture 10

BST Operations, Complexity & Traversal

Begin AVL Trees

2/20/24, Tuesday

Oregon State University

Odds and Ends

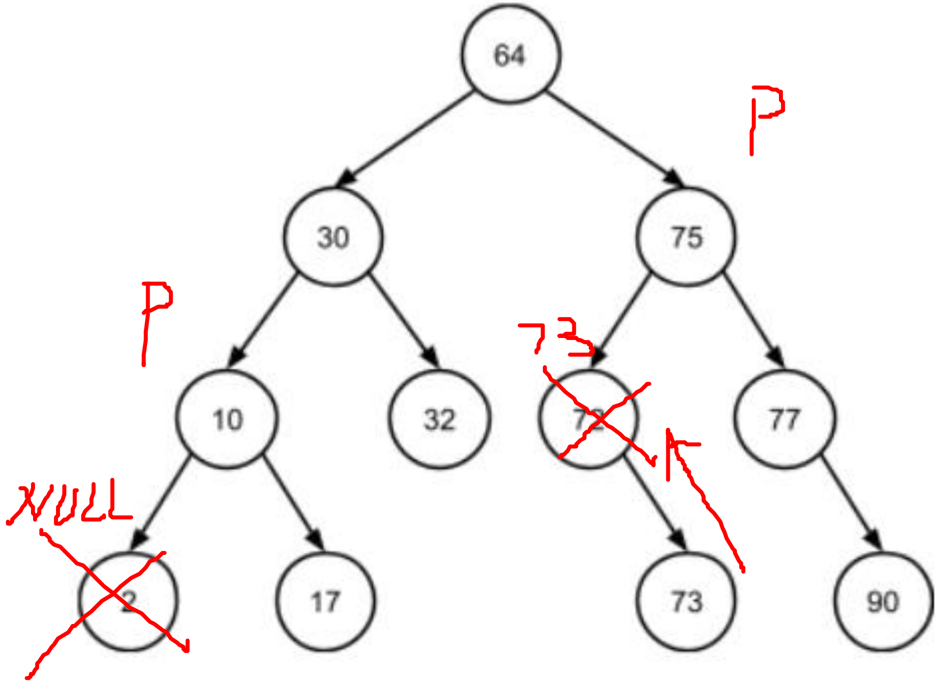
- Recitation 7 posted
- Don’t forget to demo your assignment 2!

Lecture Topics:

- BST Operations:
  - Removing an element
- Runtime Complexity of BST operations
- BST traversals

BST Operations: Removing an element

- BST removal: depend on the number of children that element's BST node has
- If the element to be removed is a leaf node: (i.e., 2)
  - simply free that node and update its parent to have a NULL child
- If the element to be removed is stored in a node with just a single child: (i.e., 72)
  - simply free that node and move its child to become a child of the node's parent

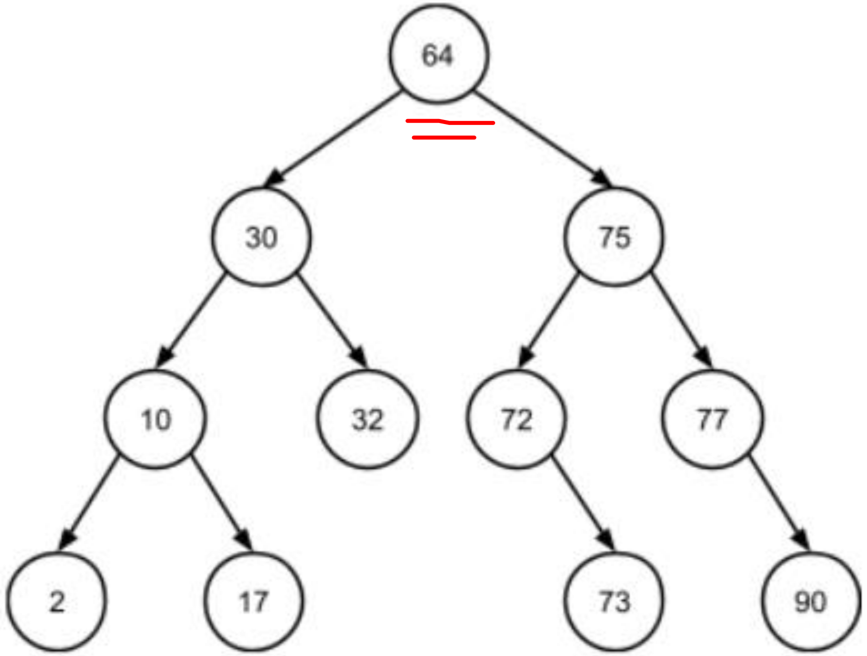


BST Operations: Removing an element

- If the element to be removed is stored in a node with two children: (i.e., 64):
  - need to find that node's ***in-order successor*** (the next node in in-order traversal of the BST).

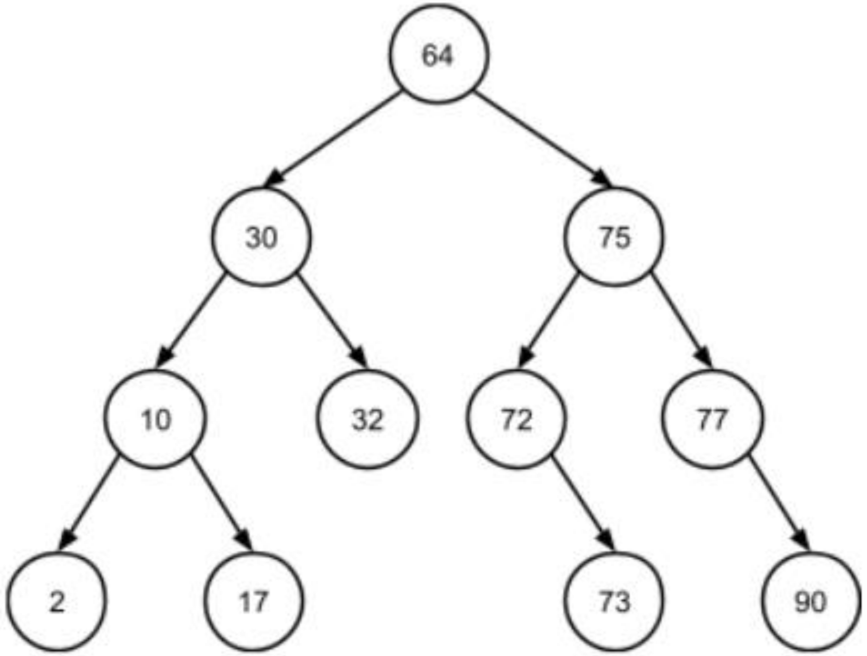
  - Line up all keys in ascending order:
  - 2 10 17 30 32 64 72 73 75 77 90

  - The in-order successor for a node with key k, is the node to the very next key after k in this ordered list of keys
    - i.e., the in-order successor of root (64) is the node with key 72



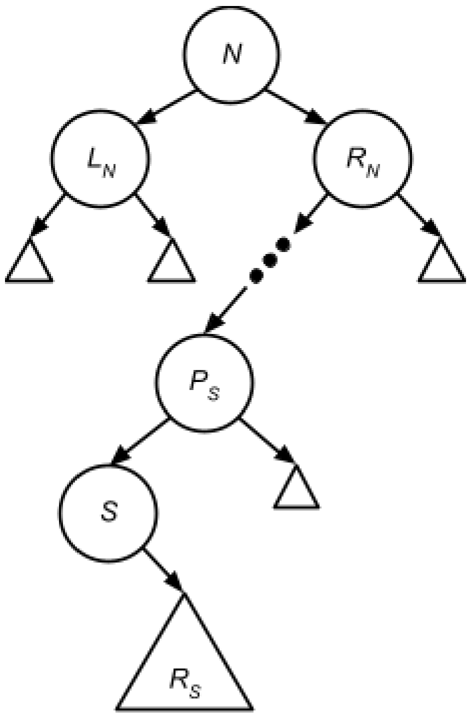
BST Operations: Removing an element

- If the element to be removed is stored in a node with two children: (i.e., 64):
  - In BST, a node N's in-order successor is always **the leftmost node in N's right subtree**.
    - branch right in the tree from N, and then continue to branch left until we can no longer do so, The last node we reach will be N's in-order successor



BST Operations: Removing an element

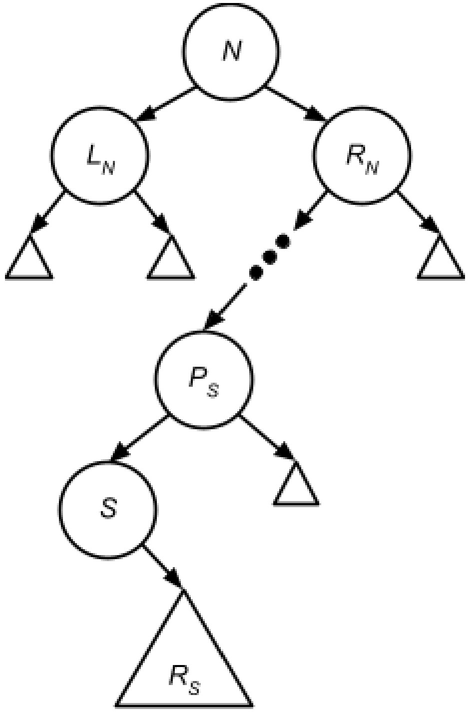
- If the element to be removed is stored in a node with two children: (i.e., 64):
  - *Denote N's parent node as P_N (if N is the root node, P_N will represent the root pointer for the entire tree)*
  - *Find N's in-order successor S. Denote S's parent node as P_S.*
  - *Update pointers to give N's children to S*
    - *N's left child becomes S's left child.*
    - *S's right child (which might be NULL) becomes P_S's left child.*
    - *N's right child becomes S's right child.*
    - *Update P_N to replace N with S.*
      - *Specifically, S becomes P_N's left or right child, as appropriate, or the root of the tree, if N was the root.*
  - *Free the node N.*



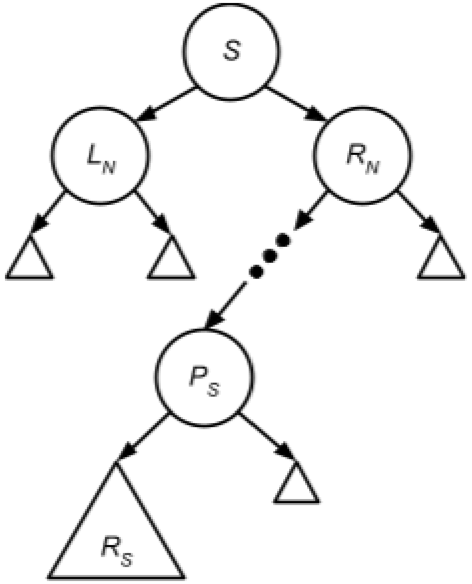
Before removing *N*

BST Operations: Removing an element

- If the element to be removed is stored in a node with two children: (i.e., 64):



Before removing *N*



After removing *N*

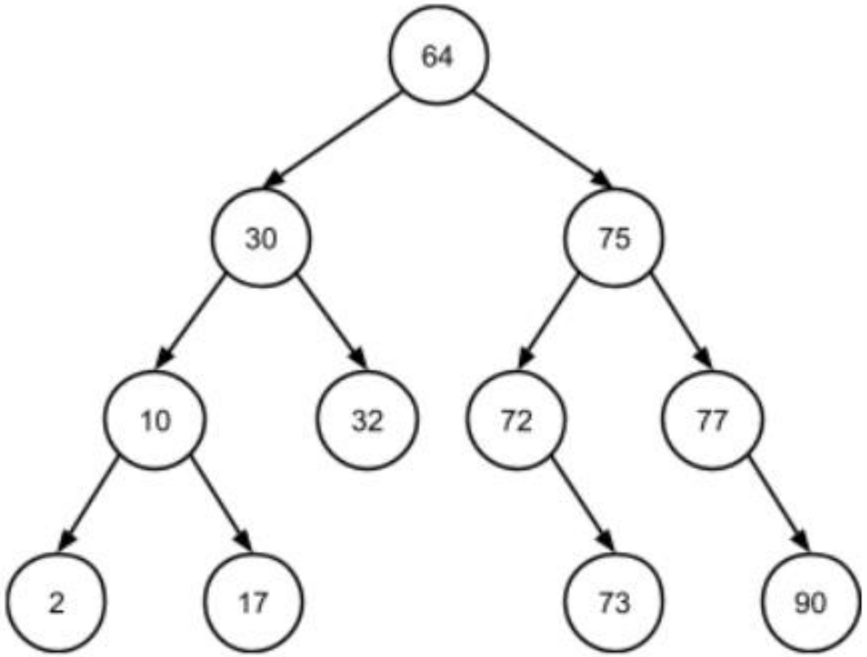
BST Operations: Removing an element

- Pseudocode:

```
remove(bst, k):
	N, P_N ← find the node to be removed and its parent
	         based on key k, as in the find() function
	if N has no children:
		update P_N to point to NULL instead of N
	else if N has one child:
		update P_N to point to N's child instead of N
	else:
		S, P_S ← find N's in-order successor and its
		         parent, as described above
		S.left ← N.left
		if S is not N.right:
			P_S.left ← S.right
			S.right ← N.right
		update P_N to point to S instead of N
	free N
```

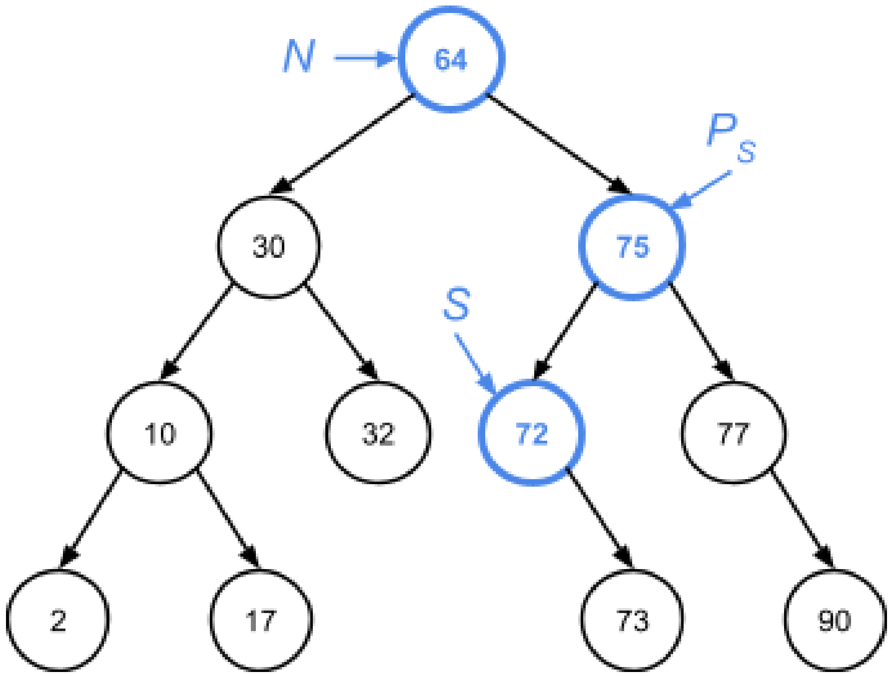
BST Operations: Removing an element

- Example: Remove the root node (64)



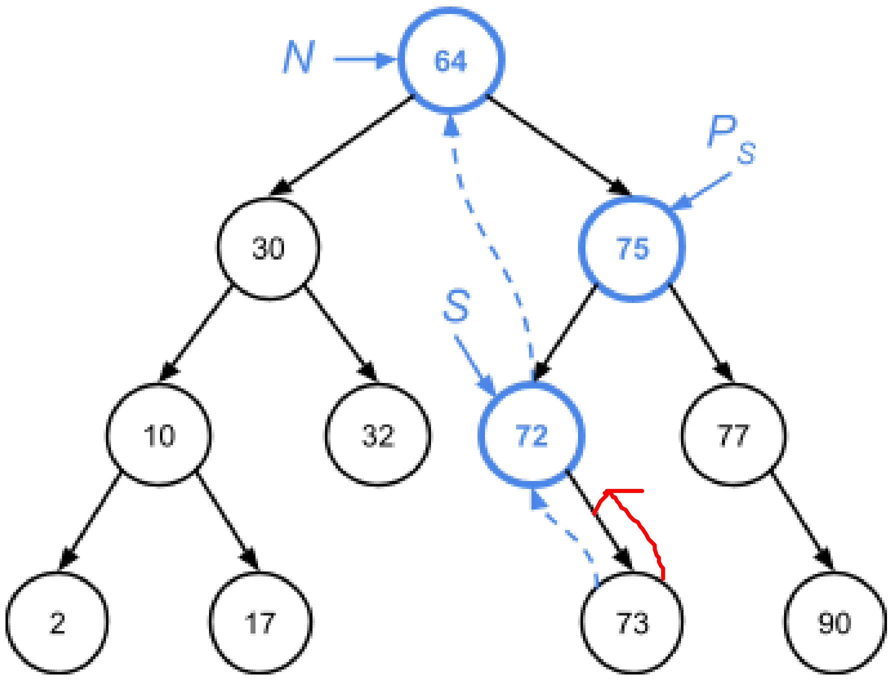
BST Operations: Removing an element

- Example: Remove the root node (64)
  - 1. identify that node's in-order successor (S) and its parent (P_S):



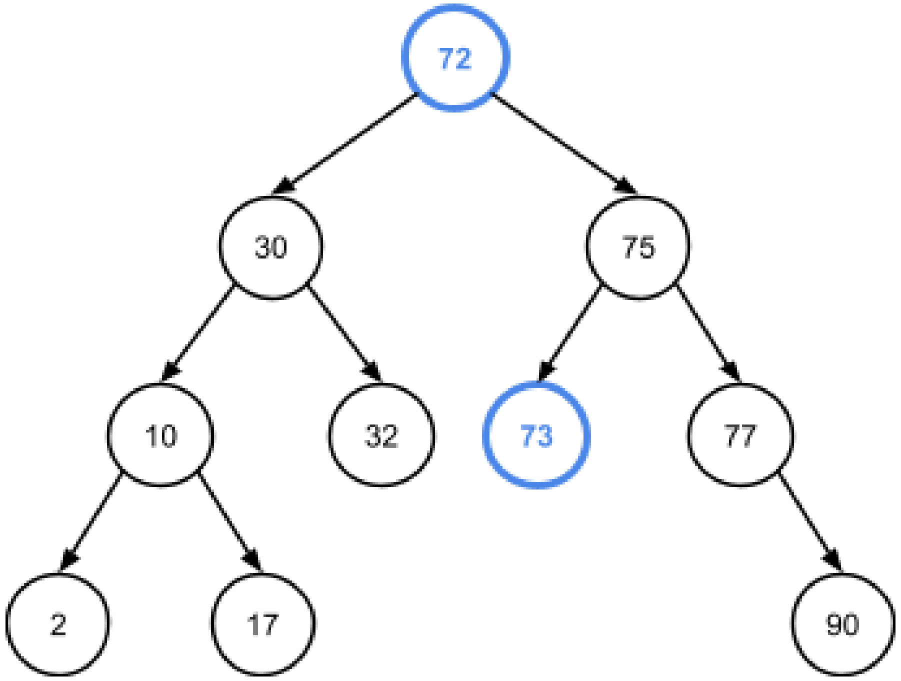
BST Operations: Removing an element

- Example: Remove the root node (64)
  - 2. update pointers so that *S* replaces *N* and *S*'s right child replaces *S* as P_S's child:



BST Operations: Removing an element

- Example: Remove the root node (64)
  - 3. The end result is a tree with the root node (i.e. N) removed.



- note that the BST property is maintained by this removal:

Lecture Topics:

- BST Operations:
  - Finding an element
  - Inserting a new element
  - Removing an element

- Runtime Complexity of BST operations
- BST traversals

Runtime Complexity of BST Operations

- Main factor of all 3 BST operations: search within the tree
  - find(): search for the query key
  - insert(): search for the location at which to insert
  - remove(): search for both query key and its in-order successor

- Search begins at the root, moves down one level at each iteration, until reaches the bottom (or finds the node it is searching for)
  - Number of search iteration == the height of the tree, h

- Thus, runtime complexity for searching in all 3 operations: O(h)

Runtime Complexity of BST Operations

- Extra work done besides searching:
  - find(): none
  - insert(): allocate the new node, and update its new parent → O(1)
  - remove(): update a few pointers → O(1)

- Thus, the runtime complexity:
  - find() – O(h)
  - insert() – O(h)
  - remove() – O(h)

- What is the range of h if the BST has n nodes?
  - Depending on the order of insertion, h can be [log(n), n]

→ limit the height of the BST! (more later)

Lecture Topics:

- BST Operations:
  - Finding an element
  - Inserting a new element
  - Removing an element

- Runtime Complexity of BST operations
- BST traversals

Binary Tree Traversal

- How to print the value stored at each node in a binary tree?

- **A tree traversal**: a method for visiting each node in a tree exactly once and performing some operation or processing at each node when it's visited

Binary Tree Traversal

- Two types of tree traversal:
  - **Depth-first**: explores a tree subtree by subtree, visiting all of a node's descendants before visiting any of its siblings.
    - moves as far downward in the tree as it can go before moving across in the tree
  - **Breadth-first**: explores a tree level by level, visiting every node at a given depth in the tree before moving downward
    - moves as far across the tree as it can go before moving down in the tree

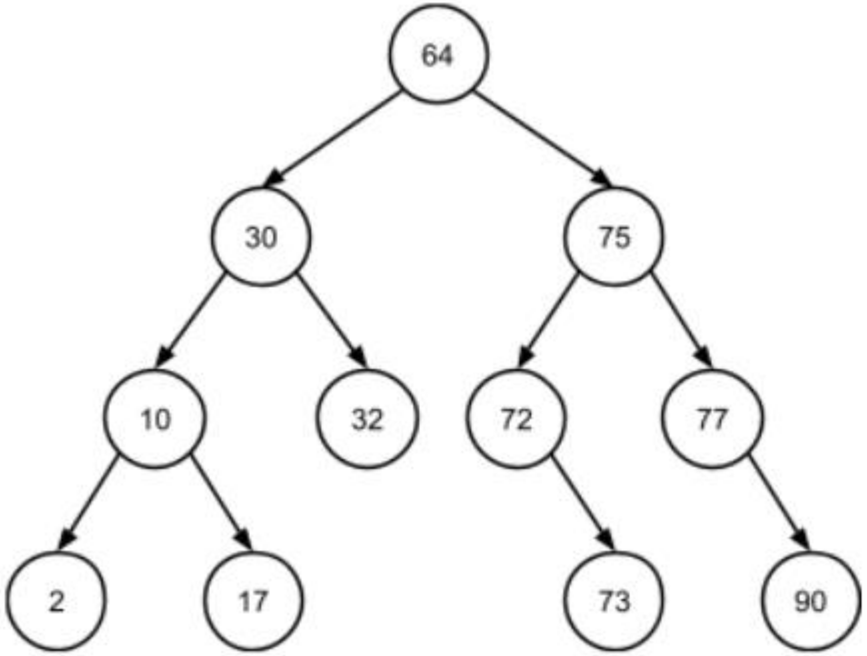
Binary Tree Traversal: Depth-first

- Denote using N, L, and R:
  - N – visit/process the current node itself
  - L – traverse the left subtree of the current node
  - R – traverse the right subtree of the current node

- Three kinds of depth-first traversal:
  - Pre-order traversal (NLR): process the current node before traversing either of its subtrees
  - In-order traversal (LNR): traverse the current node’s left subtree before processing the node itself, and then traverse the node’s right subtree
  - Post-order traversal (LRN): traverse both of the current node’s subtrees (left, then right) before processing the node itself

Binary Tree Traversal: Depth-first

- Three kinds of depth-first traversal:
  - Pre-order traversal (NLR)
    - 64 30 10 2 17 32 75 72 73 77 90
  - In-order traversal (LNR)
    - 2 10 17 30 32 64 72 73 75 77 90
  - Post-order traversal (LRN)
    - 2 17 10 32 30 73 72 90 77 75 64



- Note: in-order traversal processes the nodes in sorted order!

Binary Tree Traversal: Depth-first

- Pseudocode of three kinds of depth-first traversal: using recursion
  - Pre-order traversal (NLR)

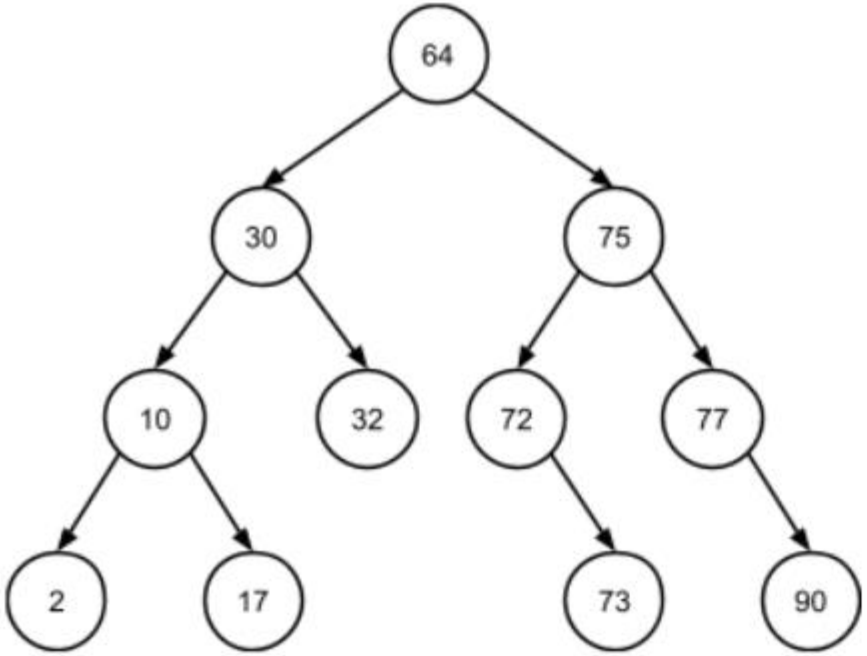
```
preOrder(N):
    if N is not NULL:
        process N
        preOrder(N.left)
        preOrder(N.right)
```

  - In-order traversal (LNR)

```
inOrder(N):
    if N is not NULL:
        inOrder(N.left)
        process N
        inOrder(N.right)
```

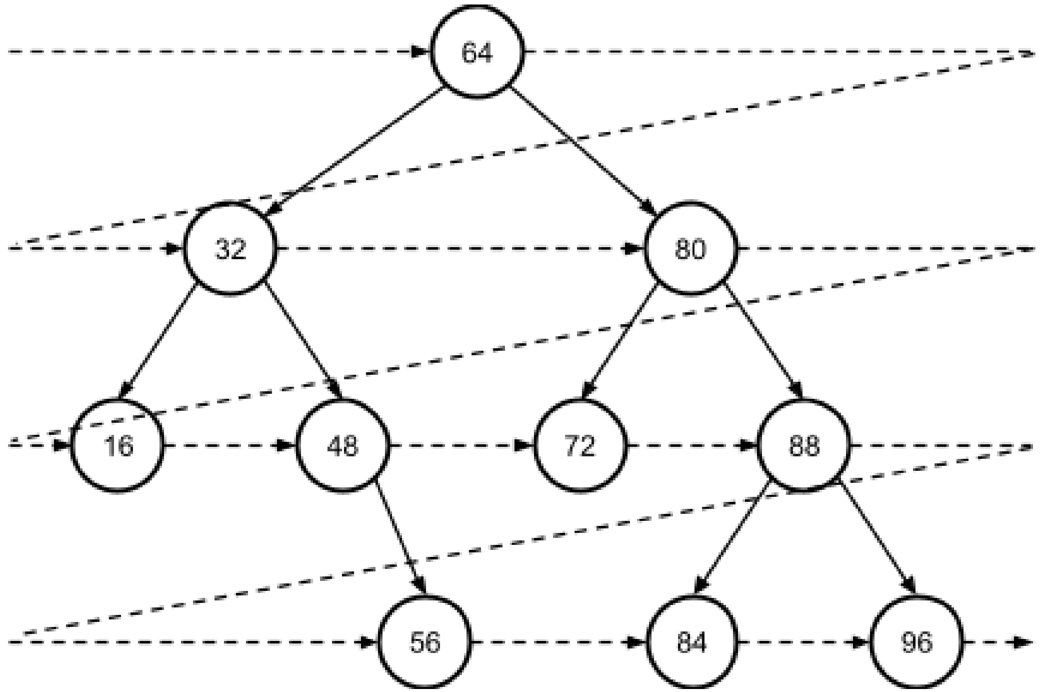
  - Post-order traversal (LRN)

```
postOrder(N):
    if N is not NULL:
        preOrder(N.left)
        preOrder(N.right)
        process N
```



Binary Tree Traversal: Breadth-first

- One main kind of breadth-first traversal: level-order traversal

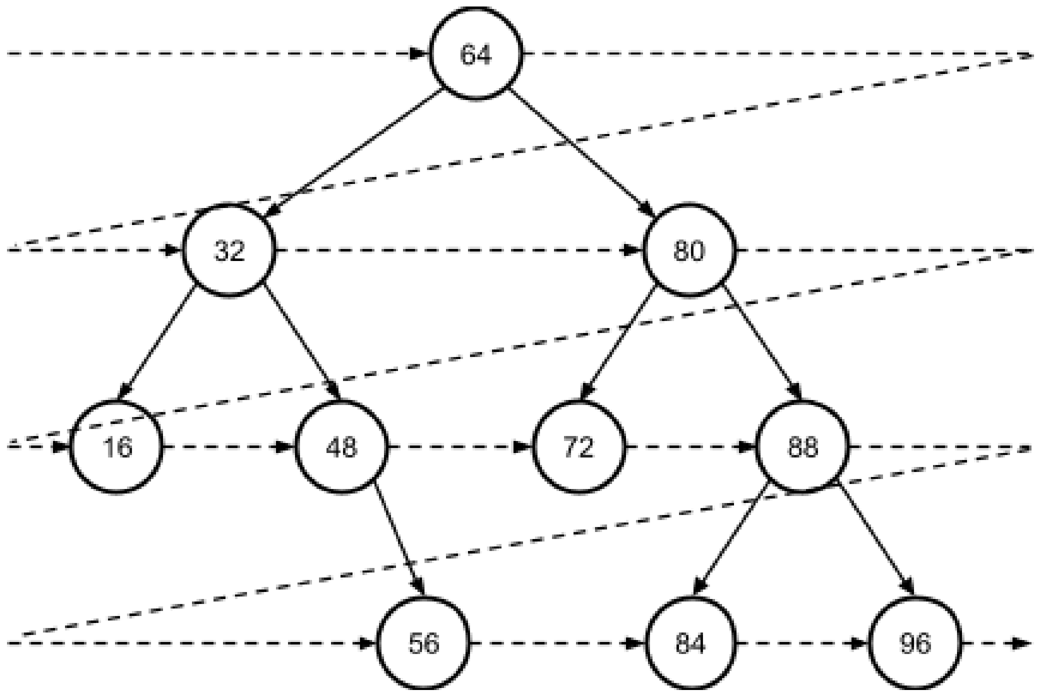


- Using a level-order traversal, the nodes are processed in this order: 64, 32, 80, 16, 48, 72, 88, 56, 84, 96.

Binary Tree Traversal: Breadth-first

- Pseudocode of level-order traversal: using a queue

```
levelOrder(bst):
    q = new, empty queue
    enqueue(q, bst.root)
    while q is not empty:
            N = dequeue(q)
            if N is not NULL:
                    process N
                    enqueue(q, N.left)
                    enqueue(q, N.right)
```



Lecture Topics:

- AVL Trees
  - Self-balancing BST

The Balance of BSTs

- Balance of BSTs:
  - All nodes have depths approximately log(n) or less

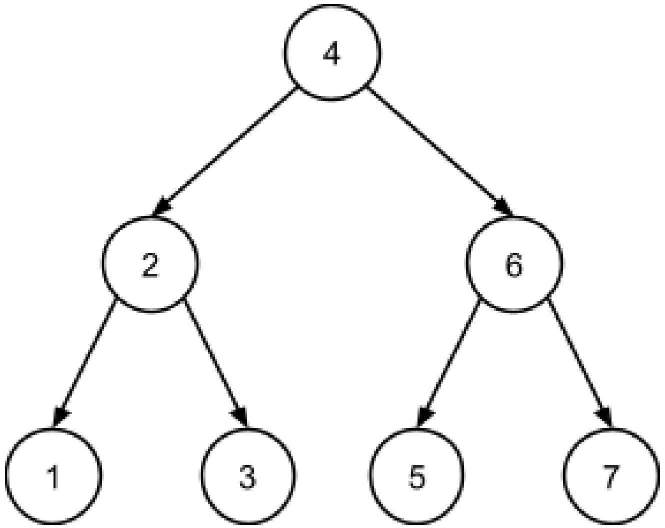
- Balance is important – primary operations on BSTs all have O(h) runtime complexity, where h is the height of the tree.

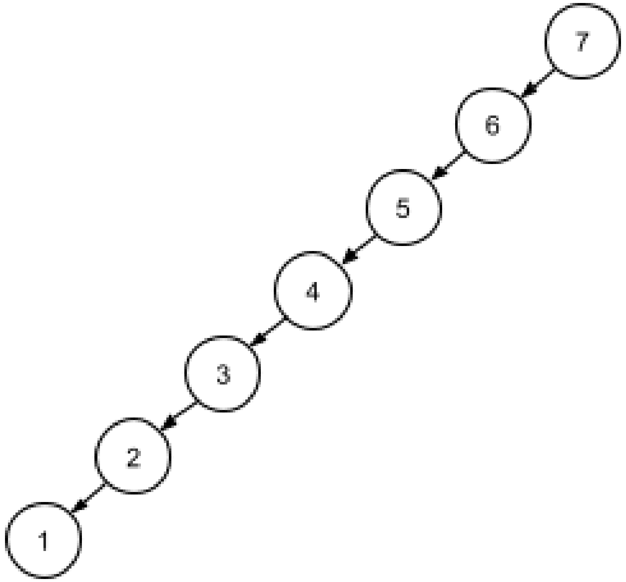
- With balanced BST, h → log(n), then O(h) will be fast
- With unbalanced BST, h → n, then O(h) will be slow

- Problem: plain BSTs cannot ensure itself is balanced

The Balance of BSTs

- Exercise:
  - Create a BST by inserting the elements with the following keys (order as they are):
    - 4 2 6 1 3 5 7
    - 7 6 5 4 3 2 1

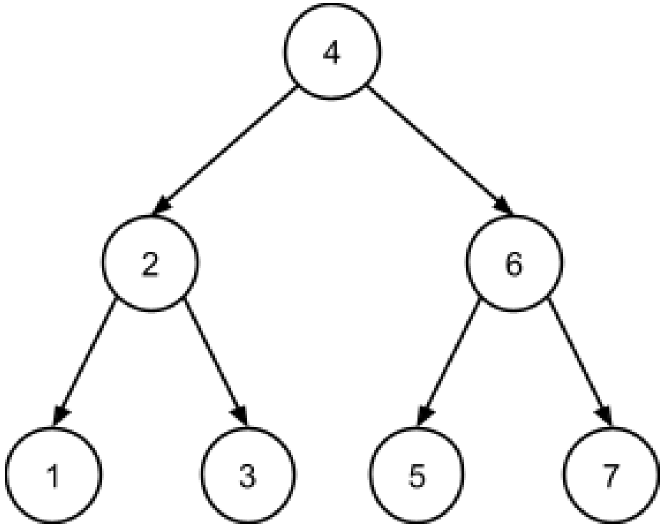


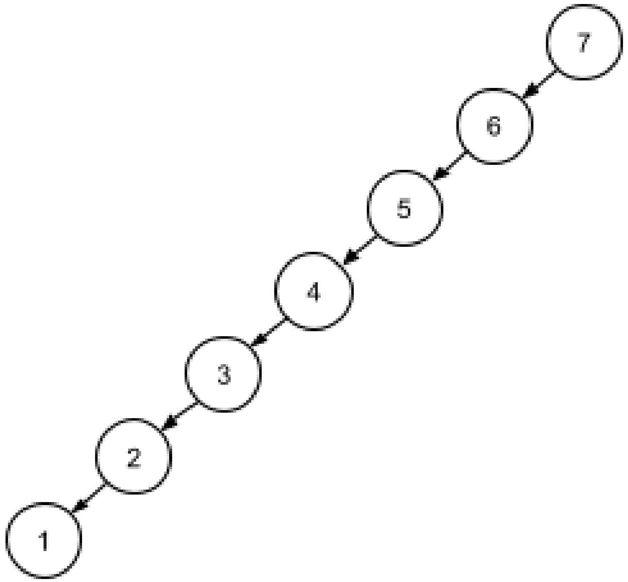


  - What do you notice?

The Balance of BSTs

- For a given set of keys, the shape of a BST depends on the order in which those keys are inserted into the tree.
  - Left: perfectly balanced, operations runtime close to O (log n)
  - Right: very unbalanced, operations runtime close to O (n)





The Balance of BSTs

- *Self-balancing BST*: does "extra work" to ensure that the tree is more-or-less balanced as elements are inserted and removed.
  - *Extra work – beyond that done by a plain BST

- A typical type of self-balancing BST known as an ***AVL tree***

Height Balance

- *Height Balance*: a measurable form of BST balance

- A BST is height balanced if, at every node in the tree, the subtree heights of the node's left and right subtrees differ by at most 1

- A height-balanced BST is guaranteed to have an overall height that's within a constant factor of log(n)
  - operations in a height-balanced BST are guaranteed to have *O(log n)* runtime complexity.

Balance Factor

- A BST node's *balance factor* – a metric to figure out whether the subtree rooted at that node is height balanced.

- the balance factor of the node N:
  - **`balanceFactor(N) = height(N.right) - height(N.left)`**

  - the height of a NULL node (i.e. an empty subtree) is -1

Balance Factor

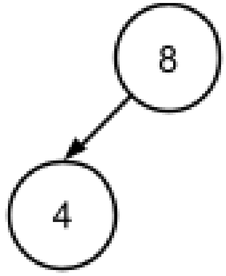
- An entire BST is *height balanced* if every node in the tree has a balance factor of -1, 0, or 1

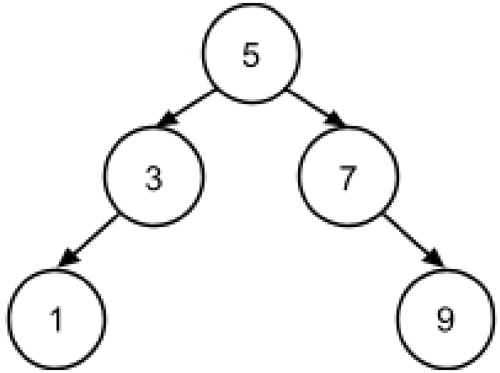
- If a node has a negative balance factor (i.e. balanceFactor(N) < 0), we call it *left-heavy*

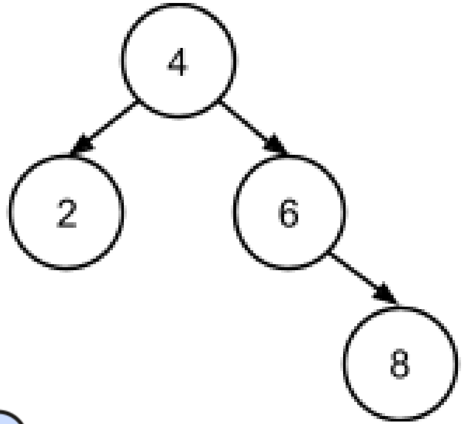
- If a node has a positive balance factor (i.e. balanceFactor(N) > 0), we call it *right-heavy*

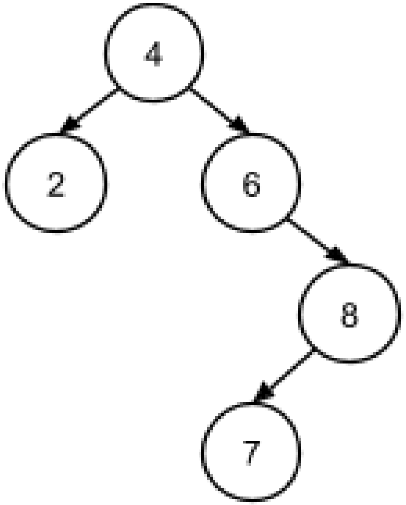
Height Balance and Balance Factor

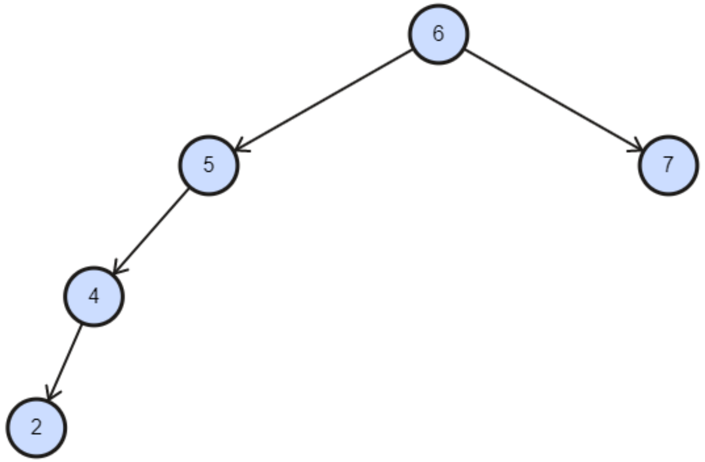
- Height-balanced, or un-balanced? Write down balance factor for each node.











Restructuring AVL Trees via Rotations

- The AVL tree is one of several existing types of self-balancing BST.
  - AVL is derived from the initials of the names of the tree's inventors: Adelson-Velsky and Landis.
  - Another popular one is the red-black tree.

- An AVL tree's operations include mechanisms to ensure that the tree always exhibits height balance
  - check the height balance of the tree after each insertion and removal
  - perform rebalancing operations known as *rotations* whenever height balance is lost